Build Script Interaction with TeamCity

If TeamCity doesn't support your testing framework or build runner out of the box, you can still avail yourself of many TeamCity benefits by customizing your build scripts to interact with the TeamCity server. This makes a wide range of features available to any team regardless of their testing frameworks and runners. Some of these features include displaying real-time test results and customized statistics, changing the build status, and publishing artifacts before the build is finished.
The build script interaction can be implemented by means of:

- service messages in the build script
- `teamcity-info.xml` file (obsolete approach, consider using service messages instead)

> If you use MSBuild build runner, you can use MSBuild Service Tasks.

In this section:

Service Messages

Service messages are specially constructed pieces of text that are used to pass commands/information about the build from the build script to the TeamCity server.

To be processed by TeamCity, they need to be written to the standard output stream of the build, i.e. printed or echoed from a build step. It is recommended to output a single service message per line (i.e. divide messages with newline symbol(s))

Examples:

Windows

```
echo ##teamcity[<messageName> 'value']
```

Linux

```
echo "##teamcity[<messageName> 'value']"
```

PowerShell script

```
Write-Host "##teamcity[<messageName> 'value']"
```

A single service message cannot contain a newline character inside it, it cannot span across multiple lines.

Service messages formats

Service messages support two formats:

- Single-attribute message:

  ```
  ##teamcity[<messageName> 'value']
  ```

- Multiple-attribute message:

  ```
  ##teamcity[<messageName> name1='value1' name2='value2']
  ```

  Multiple attributes message can more formally be described as:

  ```
  ##teamcity[messageNameWSPpropertyNameOWSP=OWSP'value'WSPpropertyName_IDOWSP=
  OWSP'value'...OWSP]
  ```

  where:

  - messageName is a name of the message. See below for supported messages. The message name should be a valid Java id (only alpha-numeric characters and "-", starting with an alpha character)
  - propertyName is a name of the message attribute. Should be a valid Java id.
  - value is a value of the attribute. Should be an escaped value (see below).
  - WSP is a required whitespace(s): space or tab character (\t)
  - OWSP is an optional whitespace(s)
  - ... is any number of WSPpropertyNameOWSP=OWSP'_value'_ blocks

Escaped values

For escaped values, TeamCity uses a vertical bar "|" as an escape character. In order to have certain characters properly interpreted by the TeamCity server, they must be preceded by a vertical bar. For example, the following message:

```
##teamcity[testStarted name='foo|'s test']
```

will be displayed in TeamCity as 'foo's test'. Please, refer to the table of the escaped values below.

Character	Escape as
' (apostrophe)	|'
\n (line feed)	|n

\r (carriage return)	|r
\uNNNN (unicode symbol with code 0xNNNN)	|0xNNNN
| (vertical bar)	||
[(opening bracket)	|[
] (closing bracket)	|]

Common Properties

Any "message and multiple attribute" message supports the following list of optional attributes: `timestamp`, `flowId`.
In the following examples `<messageName>` is the name of the specific service message.

Message Creation Timestamp

```
##teamcity[<messageName> timestamp='timestamp' ...]
```

The timestamp format is `yyyy-MM-dd'T'HH:mm:ss.SSSZ` or `yyyy-MM-dd'T'HH:mm:ss.SSS` according to Java SimpleDateFormat syntax, e.g.

```
##teamcity[<messageName> timestamp='2008-09-03T14:02:34.487+0400' ...]
##teamcity[<messageName> timestamp='2008-09-03T14:02:34.487' ...]
```

For .NET DateTimeOffset, a code like this

```
var date = DateTimeOffset.Now;
var timestamp = $"{date:yyyy-MM-dd'T'HH:mm:ss.fff}{date.Offset.Ticks:+;-;}{date.Offset:hhmm}";
```

will result in

```
##teamcity[<messageName> timestamp='2008-09-03T14:02:34.487+0400' ...]
```

Message FlowId

The flowId is a unique identifier of the messages flow in a build. Flow tracking is necessary, for example, to distinguish separate processes running in parallel. The identifier is a string that should be unique in the scope of individual build.

```
##teamcity[<messageName> flowId='flowId' ...]
```

Reporting Messages For Build Log

You can report messages for a build log in the following way:

```
##teamcity[message text='<message text>' errorDetails='<error details>' status='<status value>']
```

where:

- The `status` attribute may take following values: `NORMAL`, `WARNING`, `FAILURE`, `ERROR`. The default value is `NORMAL`.
- The `errorDetails` attribute is used only if `status` is `ERROR`, in other cases it is ignored.

This message fails the build in case its status is `ERROR` and "Fail build if an error message is logged by build runner" box is checked on the Build Failure Conditions page of a build configuration. For example:

```
##teamcity[message text='Exception text' errorDetails='stack trace' status='ERROR']
```

Blocks of Service Messages

Blocks are used to group several messages in the build log.

Block opening:

The `blockOpened` system message allows the `name` attribute, you can also add a description to the to the `blockOpened` messag e:

```
##teamcity[blockOpened name='<blockName>' description='<this is the description of blockName>']
```

Block closing:

```
##teamcity[blockClosed name='<blockName>']
```

Please note that when you close the block, all inner blocks are closed automatically.

Reporting Compilation Messages

```
##teamcity[compilationStarted compiler='<compiler name>']
...
##teamcity[message text='compiler output']
##teamcity[message text='compiler output']
##teamcity[message text='compiler error' status='ERROR']
...
##teamcity[compilationFinished compiler='<compiler name>']
```

where:

- `compiler name` is an arbitrary name of the compiler performing compilation, eg. javac, groovyc and so on. Currently it is used as a block name in the build log.
- any message with status `ERROR` reported between `compilationStarted` and `compilationFinished` will be treated as a compilation error.

Reporting Tests

To use the TeamCity on-the-fly test reporting, a testing framework needs dedicated support for this feature to work (alternatively, XML Report Processing can be used).

If TeamCity doesn't support your testing framework natively, it is possible to modify your build script to report test runs to the TeamCity server using service messages. This makes it possible to display test results in real-time, make test information available on the Tests tab of the Build Results page.

Supported test service messages

Test suite messages:
Test suites are used to group tests. TeamCity displays tests grouped by suites on Tests tab of the Build Results page and in other places.

```
##teamcity[testSuiteStarted name='suiteName']
<individual test messages go here>
##teamcity[testSuiteFinished name='suiteName']
```

All the individual test messages are to appear between `testSuiteStarted` and `testSuiteFinished` (in that order) with the same `name` attributes.

Nested test reporting

Prior to TeamCity 9.1, one test could have been reported from within another test.
In the later versions, starting another test finishes the currently started test in the same "flow". To still report tests from within other tests, you will need to specify another flowId in the nested test service messages.

Test start/stop messages:

```
##teamcity[testStarted name='testName' captureStandardOutput='<true/false>']
<here go all the test service messages with the same name>
##teamcity[testFinished name='testName' duration='<test_duration_in_milliseconds>']
```

Indicates that the test "testName" was run. If the `testFailed` message is not present, the test is regarded successful, where

- duration (optional numeric attribute) - sets the test duration in milliseconds (should be an integer) to be reported in TeamCity UI. If omitted, the test duration will be calculated from the messages timestamps. If the timestamps are missing, from the actual time the messages were received on the server.
- captureStandardOutput (optional boolean attribute) - if `true`, all the standard output (and standard error) messages received between `testStarted` and `testFinished` messages will be considered test output. The default value is `false` and assumes usage of `testStdOut` and `testStdErr` service messages to report the test output.

All the other test messages (except for `testIgnored`) with the same `name` attribute should appear between the `testStarted` and `testFinished` messages (in that order).

If using Ant's echo task to output the messages, make sure to include flowId attribute with the same value in all the messages related to the same test/test suite as otherwise they will not be processed correctly.

It is highly recommended to ensure that the pair of `test suite` + `test name` is unique within the build.
For advanced TeamCity test-related features to work, test names should not deviate from one build to another (a single test must be reported under the same name in every build). Include absolute paths in the reported test names is strongly discouraged.

Ignored tests:

```
##teamcity[testIgnored name='testName' message='ignore comment']
```

Indicates that the test "`testName`" is present but was not run (was ignored) by the testing framework.
As an exception, the `testIgnored` message can be reported without the matching `testStarted` and `testFinished` messages.

Test output:

```
##teamcity[testStarted name='className.testName']
##teamcity[testStdOut name='className.testName' out='text']
##teamcity[testStdErr name='className.testName' out='error text']
##teamcity[testFinished name='className.testName' duration='50']
```

The `testStdOut` and `testStdErr` service messages report the test's standard and error output to be displayed in the TeamCity UI. There must be only one `testStdOut` and one `testStdErr` message per test.
An alternative but a less reliable approach is to use the `captureStandardOutput` attribute of the `testStarted` message.

Test result:

```
##teamcity[testStarted name='MyTest.test1']
##teamcity[testFailed name='MyTest.test1' message='failure message' details='message and stack
trace']
##teamcity[testFinished name='MyTest.test1']

##teamcity[testStarted name='MyTest.test2']
##teamcity[testFailed type='comparisonFailure' name='MyTest.test2' message='failure message'
details='message and stack trace' expected='expected value' actual='actual value']
##teamcity[testFinished name='MyTest.test2']
```

Indicates that the "testname" test failed. Only one `testFailed` message can appear for a given test name.

- `message` contains the textual representation of the error
- `details` contains detailed information on the test failure, typically a message and an exception stacktrace.
- `actual` and `expected` attributes can only be used together with `type='comparisonFailure` to report comparison failure. The values will be used when opening the test in the IDE.

Here is a longer example of test reporting with service messages:

```
##teamcity[testSuiteStarted name='suiteName']
##teamcity[testSuiteStarted name='nestedSuiteName']
##teamcity[testStarted name='package_or_namespace.ClassName.TestName']
##teamcity[testFailed name='package_or_namespace.ClassName.TestName' message='The number
should be 20000' details='junit.framework.AssertionFailedError: expected:<20000> but
was:<10000>|n|r    at junit.framework.Assert.fail(Assert.java:47)|n|r    at
junit.framework.Assert.failNotEquals(Assert.java:280)|n|r...']
##teamcity[testFinished name='package_or_namespace.ClassName.TestName']
##teamcity[testSuiteFinished name='nestedSuiteName']
##teamcity[testSuiteFinished name='suiteName']
```

Interpreting test names

A full test name can have a form of: `<suite name>:<package/namespace name>.<class name>.<test method>(<test parameters>),`

where <class name> and <test method> can have no dots in the names. Only <test method> is a mandatory part of a full test name.

The full test name is used to compare tests between consequent builds or match tests across different build configurations.

An example of a full test name is

```
Integration Tests: Backend: org.jetbrains.teamcity.LoginPageController.testBadPassword("incorrect
password", false)

// in the example above,
// suite name = "Integration Tests: Backend"
// package = org.jetbrains.teamcity
// class name = LoginPageController
// test method = testBadPassword
// test parameters = ("incorrect password", false)
```

The Tests tab of the Build Results page allows grouping by suites, packages/namespaces, classes, and tests. Usually the attribute values are provides as they are reported by your test framework and TeamCity is able to interpret test names correctly.

If a test cannot be parsed in the form above, TeamCity still tries to extract <suite name> from the full test name for the filtering on the Tests tab, and treats everything after the suite a non-parsable test name.

Reporting additional test data

Since TeamCity 2018.2, it is possible to attach extra information to the tests, using a `testMetadata` service message.

More details on this is available on a separate page.

Reporting .NET Code Coverage Results

You can configure .NET coverage processing by means of service messages. To learn more, refer to Manually Configuring Reporting Coverage page.

Reporting Inspections

You can report inspections from a custom tool to TeamCity using the service messages described below.

Among other uses, the number of inspections can be used as a build metric to fail a build on.

Inspection type

Each specific warning or an error in code (inspection instance) has an inspection type - the unique description of the conducted inspection, which can be reported via

```
##teamcity[inspectionType id='<id>' name='<name>' description='<description>'
category='<category>']
```

where all the attributes are required and can have either numeric or textual values

`id` - (mandatory) limited by 255 characters
name - (mandatory) limited by 255 characters
`category` - (mandatory) limited by 255 characters. The `category` attribute examples are "Style violations", "Calling contracts" etc.
`description` (mandatory) limited by 4000 characters. The description can also be in html, e.g.

```
<html>
<body>
Reports unnecessary local variables, which add nothing to the comprehensibility of a method. Variables
caught include local variables which are immediately returned, local variables that are immediately
assigned to another variable and then not used, and local variables which always have the same value
as another
local variable or parameter.
<!-- tooltip end -->
<p>
Use the first checkbox below to have this inspection ignore variables which are immediately returned
or thrown. Some coding styles suggest using such variables for clarity and ease of debugging.
<p>
Use the second checkbox below to have this inspection ignore variables which are annotated.
<p>
</body>
</html>
```

Inspection instance

Reports a specific defect, warning, error message. Includes location, description, and various optional and custom attributes.

```
##teamcity[inspection typeId='<inspection type identity>' message='<instance description>'
file='<file path>' line='<line>' additional attribute='<additional attribute>']
```

where all the attributes can have either numeric or textual values:
`typeId` - (mandatory), reference to the `inspectionType.id` described above limited by 255 characters
`message` - (optional) current instance description limited by 4000 characters
`file` - (mandatory) file path limited by 4000 characters. The path can be absolute or relative to the checkout directory
`line` - (optional) line of the file, integer
`additional attribute` - can be any attribute, `SEVERITY` is often used here, with one of the following values (⚠ mind the upper case): `INFO`, `ERROR`, `WARNING`, `WEAK WARNING`

Example

```
##teamcity[inspectionType id='UnnecessaryLocalVariable' name='Redundant local variable'
description='<html><body>Reports unnecessary local variables...</body>
</html>' category='Data flow issues']
##teamcity[inspection typeId='UnnecessaryLocalVariable' message='Local variable <code>i</code>
is redundant' file='src/Test.java' line='19' SEVERITY='WARNING']
```

Publishing Artifacts while the Build is Still in Progress

You can publish the build artifacts while the build is still running, immediately after the artifacts are built.

To do this, you need to output the following line:

```
##teamcity[publishArtifacts '<path>']
```

The `<path>` has to adhere to the same rules as the Build Artifact specification of the Build Configuration settings. The files matching the `<path>` will be uploaded and visible as the artifacts of the running build.

The message should be printed after all the files are ready and no file is locked for reading.

Artifacts are uploaded in the background, which can take time. Please make sure the matching files are not deleted till the end of the build (e.g. you can put them in a directory that is cleaned on the next build start, in a temp directory or use Swabra to clean them after the build.)

⚠ The process of publishing artifacts process can affect the build because it consumes network traffic and some disk/CPU resources (should be pretty negligible for not large files/directories).

Artifacts that are specified in the build configuration setting will be published as usual.

Reporting Build Progress

You can use special progress messages to mark long-running parts in a build script. These messages will be shown on the projects dashboard for the corresponding build and on the Build Results page.

To log a single progress message, use:

```
##teamcity[progressMessage '<message>']
```

This progress message will be shown until another progress message occurs or until the next target starts (in case of Ant builds).

If you wish to show a progress message for a part of a build only, use:

```
##teamcity[progressStart '<message>']
...some build activity...
##teamcity[progressFinish '<message>']
```

⚠ The same message should be used for both `progressStart` and `progressFinish`. This allows nesting progress blocks. Also note that in case of Ant builds, progress messages will be replaced if an Ant target starts.

Reporting Build Problems

To fail a build directly from the build script, a build problem has be reported. Build problems appear on the Build Results page and also affect the build status text.
To add a build problem to a build, use:

```
##teamcity[buildProblem description='<description>' identity='<identity>']
```

where:

- `description` - (mandatory) a human-readable plain text describing the build problem. By default, the `description` appears in the build status text and in the list of build's problems. The text is limited to 4000 symbols, and will be truncated if the limit is exceeded.
- `identity` - (optional) a unique problem id. Different problems must have different identity, same problems - same identity, which should not change throughout builds if the same problem occurs, e.g. the same compilation error. It should be a valid Java id up to 60 characters. If omitted, the `identity` is calculated based on the `description` text.

Reporting Build Status

TeamCity allows changing the build status text from the build script. Unlike progress messages, this change persists even after a build has finished.
You can also change the build status of a failing build to success.

Prior to TeamCity 7.1, this service message could be used for changing the build status to failed. In the later TeamCity versions, the buildProblem service message is to be used for that.

To set the status and/or change the text of the build status (for example, note the number of failed tests if the test framework is not supported by TeamCity), use the `buildStatus` message with the following format:

```
##teamcity[buildStatus status='<status value>' text='{build.status.text} and some aftertext']
```

where:

- `status` attribute is optional and may take the value `SUCCESS`.
- `text` attribute sets the new build status text. Optionally, the text can use `{build.status.text}` substitution pattern which represents the status, calculated by TeamCity automatically using passed test count, compilation messages and so on.

The status set will be presented while the build is running and will also affect the final build results.

Reporting Build Number

To set a custom build number directly, specify a `buildNumber` message using the following format:

```
##teamcity[buildNumber '<new build number>']
```

In the <new build number> value, you can use the `{build.number}` substitution to use the current build number automatically generated by TeamCity. For example:

```
##teamcity[buildNumber '1.2.3_{build.number}-ent']
```

Adding or Changing a Build Parameter

By using a dedicated service message in your build script, you can dynamically update build parameters of the build right from a build step (the parameters need to be defined in the Parameters section of the build configuration).
The changed build parameters will be available in the build steps following the modifying one. They will also be available as build parameters and can be used in the dependent builds via `%dep.*%` parameter references, e.g.

```
##teamcity[setParameter name='ddd' value='fff']
```

When specifying a build parameter's name, mind the prefix:

- system for system properties.
- env for environment variables.
- no prefix for configuration parameter.

Read more about build parameters and their prefixes.

Reporting Build Statistics

In TeamCity, it is possible to configure a build script to report statistical data and then display the charts based on the data. Please refer to the Customizing Statistics Charts#customCharts page for a guide to displaying the charts on the web UI. This section describes how to report the statistical data from the build script via service messages. You can publish the build statics values in two ways:

- Using a service message in a build script directly
- Providing data using the teamcity-info.xml file

To report build statistics using service messages: Specify a '`buildStatisticValue`' service message with the following format for each statistics value you want to report:

```
##teamcity[buildStatisticValue key='<valueTypeKey>' value='<value>']
```

where

- The `key` should not be equal to any of predefined keys.
- The `value` should be a positive/negative integer of up to 13 digits; float values with up to 6 decimal places are also supported.

Disabling Service Messages Processing

If you need for some reason to disable searching for service messages in the output, you can disable the service messages search with the messages:

```
##teamcity[enableServiceMessages]
##teamcity[disableServiceMessages]
```

Any messages that appear between these two are not parsed as service messages and are effectively ignored.
For server-side processing of service messages, enable/disable service messages also supports the flowId attribute and will ignore only the messages with the same flowId.

Importing XML Reports

In addition to the UI Build Feature, XML reporting can be configured from within the build script with the help of the `importDat` `a` service message.
Also, the message supports importing of the previously collected code coverage and code inspection/duplicates reports.

The service message format is:

```
##teamcity[importData type='typeID' path='<path to the xml file>']
```

To be processed, report XML files (or a directory) must be located in the checkout directory, and the path must be relative to this directory.

where `typeID` can be one of the following (see also XML Report Processing):

`typeID`	Description
Testing frameworks	
`junit`	JUnit Ant task XML reports
`surefire`	Maven Surefire XML reports
`nunit`	NUnit-Console XML reports
`mstest`	MSTest XML reports
vstest	VSTest XML reports
`gtest`	Google Test XML reports
Code inspection	
`intellij-inspections`	Since TeamCity 2017.1 IntelliJ IDEA inspection results

`checkstyle`	Checkstyle inspections XML reports
`findBugs` 2)	FindBugs inspections XML reports
`jslint`	JSLint XML reports
`ReSharperInspectCode` 1)	ReSharper `inspectCode.exe` XML reports
`FxCop` 1)	FxCop inspection XML reports
`pmd`	PMD inspections XML reports
Code duplication	
`pmdCpd`	PMD Copy/Paste Detector (CPD) XML reports
`DotNetDupFinder` 1)	ReSharper `dupfinder.exe` XML reports
Code coverage	
`dotNetCoverage` 1) 3)	XML reports generated by dotcover, partcover, ncover or ncover3

Notes:
1) only supports specific file in the `path` attribute

2) also requires the `findBugsHome` attribute specified pointing to the home directory of the installed FindBugs tool.

3) also requires the `tool='<tool name>'` service message attribute, where the `<tool name>` is either `dotcover`, `partcover`, `ncover` or `ncover3`.

If not specially noted, the report types support Ant-like wildcards in the `path` attribute.

the `verbose='true'` attribute will enable detailed logging into the build log.

the `parseOutOfDate='true'` attribute will process all the files matching the path. By default, only those updated during the build (determined by last modification timestamp) are processed. If any not matching reports are found, the "report skipped as out-of-date" message appears in the build log.

the `whenNoDataPublished=<action>` (where <action> is one of the following: `info` (default), `nothing`, `warning`, `error`) will change output level if no reports matching the path specified were found.

(deprecated, use Build Failure Conditions instead)
`findBugs`, `pmd` or `checkstyle` importData messages also take optional `errorLimit` and `warningLimit` attributes specifying errors and warnings limits, exceeding which will cause the build failure.

After the `importData` message is received, TeamCity agent starts to monitor the specified paths on the disk and imports matching report files in the background as soon as the files appear on disk.

The parsing only occurs within the build step in which the messages were received. On the step finish, the agent ensures all the present reports are processed before beginning the next step. This behavior is different from that of XML Report Processing build feature, which completes files parsing only at the end of the build.

Please ensure the report files are available after the generation process ends (the files are not deleted, nor overwritten by the build script)

To initiate monitoring of several directories or parse several types of the report, send the corresponding service messages one after another.

Only several reports of different types can be included in a build. Processing reports of several inspections or duplicates tools in a single build is not supported. See the related feature request.

Libraries reporting results via TeamCity Service Messages

Several platform-specific libraries from JetBrains and external sources are able to report the results via TeamCity Service messages.

- Service messages .NET library - .NET library for generating (and parsing) TeamCity service messages from .NET applications. See a related blog post.

- Jasmine 2.0 TeamCity reporter - support for emitting TeamCity service messages from Jasmine 2.0 reporter
- Perl TAP Formatter - formatter for Perl to transform TAP messages to TeamCity service messages
- PHPUnit 5.0 - supports TeamCity service messages for tests. For earlier PHPUnit versions, the following external libraries can be used: PHPUnit Listener 1, PHPUnit Listener 2 - listeners which can be plugged via PHPUnit's `suite.xml` to produce TeamCity service messages for tests.
- Python Unit Test Reporting to TeamCity - the package that automatically reports unit tests to the TeamCity server via service messages (when run under TeamCity and provided the testing code is adapted to use it).
- Mocha - on-the-fly reporting via service messages for Mocha JavaScript testing framework. See the related post with instructions.
- Karma - support in the JavaScript testing tool to report tests progress into TeamCity using TeamCity service messages

teamcity-info.xml

As an obsolete approach, it is also possible to have the build script collect information, generate an XML file called `teamcity-info.xml` in the root build directory. When the build finishes, this file will automatically be uploaded as a build artifact and processed by the TeamCity server.

Please note that this approach can be discontinued in the future TeamCity versions, so service messages approach is recommended instead. In case service messages does not work for you, please let us know the details and describe the case via email.

Modifying the Build Status

TeamCity has the ability to change the build status directly from the build script. You can set the status (build failure or success) and change the text of the build status (for example, note the number of failed tests if the test framework is not supported by TeamCity).

XML schema for teamcity-info.xml

It is possible to set the following information for the build:

Build number — Sets the new number for the finished build. You can reference the TeamCity-provided build number using `{build.number}`.

Build status — Change the build status. Supported values are "FAILURE" and "SUCCESS".

Status text — Modify the text of build status. You can replace the TeamCity-provided status text or add a custom part before or after the standard text. Supported `action` values are "append", "prepend" and "replace".

Example `teamcity-info.xml` file:

```
<build number="1.0.{build.number}">
  <statusInfo status="FAILURE"> <!-- or SUCCESS -->
    <text action="append"> fitnesse: 45</text>
    <text action="append"> coverage: 54%</text>
  </statusInfo>
</build>
```

It is up to you to figure out how to retrieve test results that are not supported by TeamCity and accurately add them to the `teamcity-info.xml` file.

Reporting Custom Statistics

It is possible to provide custom charts in TeamCity. Your build can provide data for such graphs using `teamcity-info.xml` file.

Providing data using the teamcity-info.xml file

This file should be created by the build in the root directory of the build. You can publish multiple statistics (see the details on the data format below) and create separate charts for each set of values.

The `teamcity-info.xml` file is to contain the code in the following format (you can combine various data in the `teamcity-info.xml` file):

```
<build>
  <statisticValue key="chart1Key" value="342"/>
  <statisticValue key="chart2Key" value="53"/>
</build>
```

The `key` should not be equal to any of predefined keys.
The `value` should be a positive/negative integer of up to 13 digits. Float values with up to 6 decimal places are also supported.

The key here relates to the key of the valueType tag used when describing the chart.

Describing custom charts

See Customizing Statistics Charts page for detailed description.